Outline

Flow of control

Accumulators

Conclusion
• Prolog has a special predicate, `not`.

• The goal `not(goal)` succeeds when `goal` fails. Given:
  
  parent(sue,noel).
  parent(chris,noel).
  parent(noel,ann).
  parent(ann,dave).

• we have the following behaviour

  `?- not(parent(sue, noel)).`

  No

  `?- not(parent(ann, noel)).`

  Yes
• Note that `not` in Prolog is different to ‘not’ in English.
• For example, the above program says nothing about Dougal and Zebedee, yet

```
?- not(parent(dougal, zebedee)).
```

Yes

• In these contexts `not` means ‘not as far as I know’.
• `not` is not part of pure logic programming, and has no (clear) declarative meaning.
To think about: What happens with programs like these?

\[ p : - \text{not}(p). \]

or

\[ p : - \text{not}(q). \]

\[ q : - \text{not}(p). \]
• not can be useful in various predicate definitions
• A set is like a list except that the order of elements is unimportant, and there are no repeated elements.
• The following predicate computes the union of two sets

```prolog
union([],S,S).
union([X|S],S1,S2):-
    member(X,S1),
    union(S,S1,S2).
union([X|S],S1,[X|S2]):-
    not(member(X,S1)),
    union(S,S1,S2).
```
• It is not what you need for the first lab!
- There is a useful predicate \( \neq \) (read **not equal**)
- \( X \neq Y \) is the same as \( \text{not}(X= Y) \):
  
  ```prolog
  ?- a \neq a.
  no
  ?- a \neq b.
  yes
  ```
• There is a deadly trap involving not
• The call
  \[ \text{?- not(parent(X,noel)).} \]
  will fail (given the above program).
• It will not find a value for \( X \) which is not one of Noel’s parents, eg. Ann!
• The same applies to \( \neq \).

• The call

\[
?- X \neq a.
\]

will always fail.

• It will not find a value for \( X \) which is not equal to \( a \)!
To understand what is going on with `not`, we need to understand `!`, or ‘cut’:

- `!` always succeeds
- Once an instance of `!` has succeeded, Prolog is committed to all choices made between the matching of the clause containing that instance of `!` and the instance of `!` itself
- This includes other declarations for proving the same clause. This is very useful as we will now see.
Example:
max1(X,Y,X):- X >= Y.
max1(X,Y,Y):- X < Y.
is (well, sort of) equivalent to
max1(X,Y,X):- X >= Y,!.
max1(X,Y,Y).

Warning: deliberate error.
• It helps to consider the search tree

max1(7, 6, Z)

7 > 6,
!

max1(X, Y, X):- X >= Y, !.
max1(X, Y, Y).

Z = 7

Z = 6

No goals

No goals
• Why does the program

\[ \text{max1}(X,Y,X) :\% X \geq Y, !. \]
\[ \text{max1}(X,Y,Y). \]

contain an error?

• Because all arguments might be instantiated:

\[ 2 \%- \text{max1}(3,2,X). \]

\[ X = 3 ; \]

\[ \text{No} \]
\[ 5 \%- \text{max1}(3,2,2). \]

\[ \text{Yes} \]
• How do we fix this?
• By delaying the instantiation of the third variable in the first rule:
  \[
  \text{max2}(X,Y,Z) : - X \geq Y, !, X=Z. \\
  \text{max2}(X,Y,Y).
  \]
• This produces the correct behaviour:
  18 ?- \text{max1}(3,2,2).
  Yes
  19 ?- \text{max2}(3,2,2).
  No
• Here is another example of cut:

  member(X,[X|L]).
  member(X,[Y|L]):- member(X,L).

  one_member(X,[X|L]):- !.
  one_member(X,[Y|L]):- one_member(X,L).

• This is just like ordinary `member` except that it does not find repeated solutions
• Thus:

?- member(a, [a, b, a, c]).
yes
?- member(X, [a, b, a, c]).
X = a ;
X = b ;
X = a ;
X = c ;
No
?- one_member(a,[a, b, a, c]).
Yes
?- one_member(X,[a, b, a, c]).
X = a ;
No
• The relationship between not and !:

• Note that call(term) calls term as if it were a goal in its own right.

?- ancestor(sue,N).
N = noel

Yes

?- call(ancestor(sue,N)).
N = noel

Yes
• We can define not in terms of !.
  
  not(Goal):- call(Goal), !, fail.
  not(Goal).
Outline

Flow of control

Accumulators

Conclusion
• Time to worry about efficiency
• Consider again the definition of append/3
  \[
  \text{append}([X|L1],L2,[X|L3]):- \\
  \text{append}(L1,L2,L3). \\
  \text{append}([],L,L).
  \]
• Calls to append/3 evidently involve a number of goal calls which is \texttt{linear} in the first argument.
Consider again the definition of reverse/2

\[
\text{rev1}([X|L], \text{L}_\text{ans}) :- \\
\text{rev1}(L, \text{L}_\text{ans1}), \\
\text{append}(\text{L}_\text{ans1}, [X], \text{L}_\text{ans}).
\]

\[
\text{rev1}([], []). \\
\text{where append/3 is as defined above}
\]
• Suppose the list we are reversing has $N$ elements
• For the $k$th item in the list, we perform an append on the end of an $N - k$ element list.
• This append takes $N - k + 1$ reductions
• So the whole operation takes

$$(N + 1) + N + \ldots + 1 = \frac{1}{2}(N + 1)(N + 2)$$

reductions
• That is, it is quadratic in $N$.
• Although rev1 is very easy to understand, quadratic behaviour seems unacceptable.
Here is a better program:

```prolog
rev2(L,L1):-
    rev_acc(L,[],L1).
```

```prolog
rev_acc([X|L],L_acc,L_ans):-
    rev_acc(L,[X|L_acc],L_ans).
```

```prolog
rev_acc([],L_acc,L_acc).
```

In operation:

```prolog
?- rev2([a, b, c, d, e], L).
```

L = [e, d, c, b, a]

The middle variable in `rev2` is called an **accumulator**.
• We can see better what is going on by tracing:

\[
\begin{align*}
\text{call } & \text{ rev2}([a, b, c], \_873) \\
\text{call } & \text{ rev\_acc}([a, b, c], [], \_873) \\
\text{call } & \text{ rev\_acc}([b, c], [a], \_873) \\
\text{call } & \text{ rev\_acc}([c], [b, a], \_873) \\
\text{call } & \text{ rev\_acc}([], [c, b, a], \_873) \\
\text{exit } & \text{ rev\_acc}([], [c, b, a], [c, b, a]) \\
\text{exit } & \text{ rev\_acc}([c], [b, a], [c, b, a]) \\
\text{exit } & \text{ rev\_acc}([b, c], [a], [c, b, a]) \\
\text{exit } & \text{ rev\_acc}([a, b, c], [], [c, b, a]) \\
\text{exit } & \text{ rev2}([a, b, c], [c, b, a])
\end{align*}
\]

• Obviously, the number of reductions performed by \text{rev2} is linear in the length of the list.
• Here is another way to think about the same thing.
• Now consider the predicate `append_dfl`:
  
  `append_dfl(L1/L2,L2/L3,L1/L3).`

• The time taken to query `append_dfl` is constant
• The program clearly appends lists (look at the variable L):
  
  ```
  :- append_dfl([a, b, c| X]/X, [d, e, f| Y]/Y, U/[])
  
  N1 X = [d, e, f], Y = [], U = [a, b, c, d, e, f]
  ```
• We are interested in structures of the form

\[ x_1, \ldots, x_i, x_{i+1}, \ldots, x_n / [x_{i+1}, \ldots, x_n] \]

• This represents the list

\[ [x_1, \ldots, x_i] \]

i.e. the difference between the two lists.

• We are particularly interested in difference lists such as

\[ [a, b, c \mid X] / X. \]

• Such data-structures are said to be **incomplete**
The following picture illustrates what is going on with a call to the goal `append_dfl([a, b, c|X]/X, [d, e, f|Y]/Y, U/[])` given the program `append_dfl(L1/L2, L2/L3, L1/L3).`
• We can also write a fast list-reverse program using difference lists
  
  \[
  \text{rev3}(L,L1):-
  \text{rev\_dfl}(L,L1/[]):
  \]

  \[
  \text{rev\_dfl}([X|L],L\_ans/L\_ans\_tail):-
  \text{rev\_dfl}(L,L\_ans/[X|L\_ans\_tail]).
  \]

  \[
  \text{rev\_dfl}([],L\_ans\_tail/L\_ans\_tail).
  \]

• This is just like \text{rev1}, except that the method of appending used in \text{append\_dfl} has been ‘built in’ to the definition
• The program certainly works:

?- rev3([1, 2, 3], L)

L = [3, 2, 1]

call rev3([1, 2, 3], _897)
call rev_dfl([1, 2, 3], _897/[])
call rev_dfl([2, 3], _897/[1])
call rev_dfl([3], _897/[2, 1])
call rev_dfl([], _897/[3, 2, 1])
exit rev_dfl([], [3, 2, 1]/[3, 2, 1])
exit rev_dfl([3], [3, 2, 1]/[2, 1])
exit rev_dfl([2, 3], [3, 2, 1]/[1])
exit rev_dfl([1, 2, 3], [3, 2, 1]/[])
• If this looks unfamiliar, it shouldn’t.

rev3(L,L1):-
    rev_dfl(L,L1/[]).
rev_dfl([X|L],L_ans/L_ans_tail):-
    rev_dfl(L,L_ans/[X|L_ans_tail]).
rev_dfl([],L_ans_tail/L_ans_tail).

rev2(L,L1):-
    rev_acc(L,[],L1).
rev_acc([X|L],L_acc,L_ans):-
    rev_acc(L,[X|L_acc],L_ans).
rev_acc([],L_acc,L_acc).
• As a final example, recall
  
  factorial(0,1).
  
  factorial(N,F):-
      N > 0,
      N1 is N - 1,
      factorial(N1,F1),
      F is N * F1.

• The Prolog interpreter has to store the program state on the stack before making each recursive call.
• Here is an alternative using a numerical accumulator:

factorial2(N,F):-
    factorial2_acc(N,1,F).

factorial2_acc(N,Acc,F):-
    Acc1 is Acc * N,
    N1 is N - 1,
    factorial2_acc(N1,Acc1,F).

factorial2_acc(0,Acc,Acc).

• The Prolog interpreter can forget about the program state on the stack before making each recursive call.
• These two programs require the same number of calls as each other
• Nevertheless, the second is more efficient
• This is because Prolog does not need to keep a stack of the factorial2-goals, and can reclaim the space taken up by each.
• This space reclamation is handled by the Prolog compiler, and is called **tail-recursion optimization**.
Outline

Flow of control

Accumulators

Conclusion
• **Summary:**
  - Negation: `not` and `\=`
  - Flow of control: the cut `!`
  - Accumulators
  - Incomplete data structures: difference lists

• **What should I do next?**
  - Revise Chh. 6, 10 of *Learn Prolog Now!*
  - Read Introduction and Chh. 1,5 of *Representation and inference for natural Language* for next lecture.